Amur: Fixing Multi-Resource Leaks Guided by Resource Flow Analysis

Jinyoung Kim
Department of Software
Sungkyunkwan University
Suwon, Republic of Korea
danpoong@skku.edu

Eunseok Lee*
College of Computing and Informatics
Sungkyunkwan University
Suwon, Republic of Korea
leees@skku.edu

***Abstract*—Resource leaks pose a persistent threat to software reliability, resulting in resource exhaustion, performance degradation, and system crashes. Existing automated repair approaches, primarily based on rigid templates, are limited in handling complex or multi-resource leak scenarios and often compromise program semantics. Although recent advances in Large language models show promise in program repair, existing LLM-based methods frequently generate semantically invalid patches for resource leaks. This paper presents *Amur*, a semantics-aware patching framework that leverages static analysis to guide LLMs in repairing both single- and multi-resource leaks. At the core of Amur is a novel *Resource Flow Analysis (RFA)*, a flow-sensitive and inter-resource-aware static analysis that captures resource usage patterns and dependencies. RFA identifies potential leak points and enforces semantic constraints to guide LLMs in synthesizing semantics-preserving patches. We evaluate Amur on the NJR-1 dataset and the new JLeaks benchmark, which targets realistic multi-resource leak scenarios. Amur achieves substantial improvements over state-of-the-art methods, improving patch accuracy by 33% over RLFixer and 24% over LLM-only baselines in single-resource leak cases. For multi-resource leaks, Amur generates patches in 96% of cases and achieves 80% correctness, outperforming RLFixer and LLM-only baselines by 76% and 16%, respectively. These results demonstrate that integrating RFA into LLM-guided repair significantly enhances the correctness and generalizability of automated resource leak fixes.**

***Index Terms*—Automated Repair, Resource Leak, LLM**

I. Introduction

Resource leaks degrade system performance and reliability by consuming unreleased resources, often causing resource exhaustion, reduced responsiveness, and system crashes [1], [2]. As modern software systems scale in size and complexity, even small leaks can accumulate over time and trigger cascading failures in long-running applications [3], [4]. Although automated program repair (APR) has advanced in addressing functional bugs [5]–[10], repairing resource leaks remains difficult. The challenge lies in accurately modeling resource acquisition and release across complex control flows, exception paths, and interdependent resource lifecycles [11]. These difficulties become particularly severe when developers work with large, legacy, or poorly documented codebases.

Template-based tools, such as RLFixer [12], represent the current state-of-the-art in repairing resource leaks. RLFixer statically analyzes source code and applies predefined patch templates, achieving a 66% patch generation rate and 95% correctness on the *NJR-1* benchmark [13]. However, its reliance on manually crafted templates limits its scalability and effectiveness in repairing complex, multi-resource leaks, and may result in semantic inconsistencies. In contrast, recent advances in large language models (LLMs), such as GPT-4o, offer a promising alternative for automated repair. LLMs demonstrate strong capabilities in general-purpose code generation and have shown potential in automated program repair (APR) tasks [14]–[21]. Nonetheless, they still struggle with resource-centric bugs that require coordinated handling of multiple resources [22]–[26].

Our empirical evaluation of three state-of-the-art LLMs, namely GPT-4o, GPT-4o mini, and o1, reveals frequent semantic violations, including missing or duplicated releases and incorrect control-flow changes. These models represent a strong and diverse sample of current LLM capabilities. These findings underscore a key insight: effective resource leak repair requires semantics-aware reasoning. Neither static templates nor large language models alone can fully capture the dynamic behavior of resource usage and interdependencies.

To address this limitation, we propose a hybrid approach that combines the contextual understanding of LLMs with static analysis for semantic validation. We present **Amur**, the first semantics-aware APR framework that supports both single- and multi-resource leak repair. Amur introduces a novel *Resource Flow Analysis* that tracks resource acquisition, propagation, and release along control-flow paths. This analysis detects inter-resource dependencies and guides LLM-based patch generation toward semantically correct and minimally intrusive repairs. Unlike traditional pointer analysis [27], Resource Flow Analysis(RFA) provides task-specific structure optimized for LLM-guided resource leak repair, enabling the generation of semantically correct and minimally intrusive patches.

We evaluated Amur on two benchmarks: NJR-1 [13], used in prior state-of-the-art work, and JLeaks [28], which features realistic multi-resource leak scenarios. Amur significantly outperforms both RLFixer and LLM-only baselines. For single-resource leak repair, Amur achieves an average patch generation rate of 70% and a correctness of 81%, surpassing RLFixer by up to 33% and LLM-only methods by up to 24%. On JLeaks, RLFixer fails to generate meaningful patches, while

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00252

Amur maintains high effectiveness. The benefits are even greater in multi-resource scenarios: Amur achieves a 96% generation rate and up to 80% correctness, outperforming RLFixer by 76% and LLM-only methods by 16%. Amur's performance stems not from model selection but from its integration of static semantics and resource dependency inference, enabling robust generalization across diverse and complex codebases.

Contributions. We make the following contributions:

- We empirically identify the limitations of both template-based and LLM-based APR approaches in repairing multi-resource leaks and preserving program semantics.
- We propose an approach that guides LLM-based patch generation using flow-sensitive resource analysis, enabling semantics-aware resource leak repair.
- We introduce novel *Resource Flow Analysis*, a static analysis technique that models resource lifecycles, usage contexts, and inter-resource dependencies to ensure semantically valid and minimally invasive repairs.
- We perform extensive evaluations on two public benchmarks and show that Amur outperforms state-of-the-art methods in both correctness and generalizability.

II. Motivating Examples

We analyze four real-world Java cases to evaluate the limitations of RLFixer and LLM-based repair methods for resource leaks. For the LLM-based approaches, we employ GPT-4o, GPT-4o-mini, and o1, all yielding consistent results. Our study compares the strengths and weaknesses of RLFixer and LLM-based methods across varying leak complexities, highlighting both their effectiveness and their limitations in producing safe, reliable repairs.

```
FileInputStream fis = null;
fis = new ...;
fis.read(b);
fis.close(); // Resource Leak if read()
    throws
return b;
```

Buggy Code (a)

```
FileInputStream fis = null;
try {
  fis = new ...;
  fis.read(b);
  return b;
} finally {
  if (fis != null) fis.close();
}
```

Patch by RLFixer (b)

```
try (FileInputStream fis = new ...) {
  fis.read(b);
}
return b;
```

Patch by LLM (c)

Fig. 1: Simple single resource leak and corresponding fixes by RLFixer and LLMs.

As shown in Figure 1(a), the example illustrates a simple resource leak where an exception during `fis.read(b);` bypasses the explicit `close()` call. In Figure 1(b), RLFixer fixes this using a `try-finally` block, while the LLM-based approach, In Figure 1(c), uses the `try`-with-resources construct. Both ensure proper resource cleanup and preserve the original program behavior, as expected for this straightforward case.

```
ByteArrayOutputStream bos = new ...;
ObjectOutputStream oos = new ...;
oos.writeObject(value);
oos.close(); // Resource Leak
```

Buggy Code (a)

```
try {
  ByteArrayOutputStream bos = new ...;
  ObjectOutputStream oos = new ...;
  oos.writeObject(value);
} finally {
  oos.close();
}
```

RLFixer Patch (b)

```
try (ByteArrayOutputStream bos = new ...;
    ObjectOutputStream oos = new ...;) {
  oos.writeObject(value);
}
```

LLM Patch (c)

Fig. 2: Simple multi resource leak and corresponding fixes by RLFixer and LLMs.

In Figure 2(a), presents a multi-resource leak scenario. In Figure 2(b), RLFixer fails due to its single-resource template limitation. The fix misses the closure of `bos` and may access an out-of-scope variable in the `finally` block. In contrast, the LLM-based approach in Figure 2(c) correctly applies Java try-with-resources to manage both bos and oos, showing more substantial support for multi-resource patterns.showing more substantial support for multi-resource patterns.

In Figure 3(a), illustrates a complex single-resource leak where both RLFixer and the LLM-based approach fail. The evaluation excludes RLFixer from this example onward, as it targets only simple single-source leak cases and fails to generate a patch for the given scenario. The original code calls `acceptor.accept()` to create a socket; if this call throws an exception, the program exits the `while` loop and closes the socket properly. A correct patch must preserve this behavior. However, the LLM-generated patch in Figure 3(b) moves `close()` inside the loop and suppresses the exception, breaking the intended control flow. This results in repeated looping and a persistent leak, demonstrating how incorrect exception handling can violate control-flow semantics in automated repair.

```
try (...) {
  while (true) {
    Socket socket = acceptor.accept();
    ...
  }
} catch (IOException e) {...}
```

Buggy Code (a)

```
try (...) {
  while (true) {
    Socket socket = null;
    try {
      socket = acceptor.accept();
      ...
      socket.close();
    } catch (Exception e) {...}
  }
} catch (IOException e) {...}
```

Patch by LLM (b)

Fig. 3: Complex single resource leak and corresponding fixes by LLMs

```
try {
  HttpURLConnection con = new ...;
  ...
  DataOutputStream wr = new ...;
  ...
  wr.close();
  ...
  BufferedReader in = new ...;
  ...
  in.close();
  return response.toString();
} catch (IOException e) {
  throw new RuntimeException(...);
}
```

Buggy Code (a)

```
try {
  HttpURLConnection con = new ...;
  ...
  DataOutputStream wr = new ...;
  ...
  BufferedReader in = new ...;
  ...
  return response.toString();
} catch (IOException e) {
  throw new RuntimeException(...);
} finally {
  wr.close();
  in.close();
}
```

Patch by LLM (b)

Fig. 4: Complex multi resource leak and corresponding fixes by LLMs

Figure 4(a) presents a complex multi-resource leak where both RLFixer and the LLM fail. The buggy code follows a structure where resource acquisition, usage, and release occur in a sequential and logically scoped manner. This ordering is essential to prevent resource leaks while preserving the original control and data flow semantics. However, the LLM-generated patch in Figure 4(b) fails to capture these dependencies and instead indiscriminately defers resource cleanup to the program's termination phase. As a result, it disrupts the original flow and compromises the correctness of any fault-handling or follow-up logic associated with the early release of resources. Such failures demonstrate the limitations of LLM-based repair approaches in modeling resource lifetimes and their contextual interactions.

Summary. These cases show that RLFixer can handle simple resource leaks using predefined templates but fails when resource management involves complex control flow or multiple interacting resources. Similarly, LLM-based approaches may produce plausible repairs in simple settings, but they consistently fail to preserve correctness in the presence of inter-resource dependencies or nontrivial exception-handling logic. These limitations highlight the need for semantics-aware repair techniques that can reason precisely about resource lifetimes, execution context, and exception behavior.

III. Approach

A. Overview of Amur

Figure 5 presents the two-phase workflow of Amur, which consists of *Resource Flow Analysis (RFA)* and *Prompt-Guided Patch Generation*.

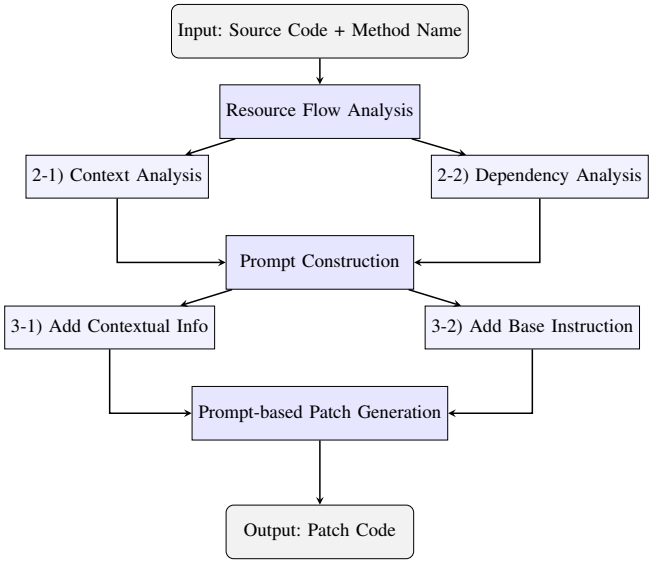

Fig. 5: Overview of the Amur, integrating resource flow analysis with LLM-guided patch generation.

Given source code and a method name, RFA performs interprocedural, flow-sensitive analysis to identify resource acquisition, propagation, and release points. This analysis captures lifecycle events, usage scopes, and inter-resource dependencies, forming the foundation of structured prompts for a Large Language Model (LLM). In the second phase, these

prompts guide the LLM to synthesize semantically consistent and functionally correct patches. The final output is a repaired version of the input code. Amur targets Java resource leaks by integrating static analysis with LLM-based patch synthesis. Unlike RLFixer, which relies on predefined templates and focuses on single-resource scenarios, Amur handles complex control flows and multiple interacting resources. Template-based methods [29] suffer from limited generalizability, while LLM-only techniques [30] often violate semantic constraints. Amur overcomes both limitations by combining precise analysis with prompt-based synthesis. Amur consists of two main components:

1. **Resource Flow Analysis (RFA)** (Section III-B): Conducts interprocedural dataflow analysis to trace resource usage across the codebase.
2. **Prompt-Based Patch Generation** (Section III-C): Converts analysis results into structured prompts to steer the LLM in producing lifecycle-aware patches.

A patch is considered correct if it satisfies all of the following:

- Eliminates the reported resource leak
- Preserves the program's original behavior
- Maintains structural consistency
- Introduces no new defects

B. Resource Flow Analysis

*1) **Definition of Resource Flow Analysis**:* The goal of Resource Flow Analysis (RFA) is to statically extract flow-sensitive, semantics-aware resource usage patterns from a method. The analysis produces two artifacts (1):

- A **Context Graph (CG)**, which partitions the method into semantically labeled code blocks and models the control-flow relationships between them.
- A **Resource Dependency Graph (RDG)**, which captures dependency relationships among resource variables, such as usage order.

$$\mathcal{RFA}(M) = (CG, RDG) \tag{1}$$

- $CG = (V, E, \tau)$ is a flow-sensitive context graph. We define this structure below.
- $RDG = (V_R, E_R)$ is a directed acyclic graph of resource variables and their semantic dependencies. Defined in Section III-B3.

*2) **Context Analysis**:* As described in the definition of RFA, Context Analysis constructs the Context Graph (CG) by decomposing a method into semantically coherent code blocks. Each block is tagged with its role in resource usage (e.g., acquire, use, release), and control flow transitions are explicitly modeled. This flow-sensitive modeling is critical for modeling correct release semantics in the presence of control constructs such as try-catch-finally, early returns, and loops. We define the output of Context Analysis as a Context Graph (CG), as shown in (2):

$$CG = (V, E, \tau) \tag{2}$$

- V is the set of *Context blocks*, where each block represents a contiguous sequence of program statements that collectively perform a coherent operation. A block is either resource-relevant (e.g., acquire, use, release) or non-resource-relevant (e.g., computation, control).
- E is the set of directed edges representing control-flow transitions between context blocks. These edges preserve the structural and execution dependencies among blocks, capturing valid execution paths.
- τ is a semantic labeling function, defined as $\tau : V \rightarrow \{\mathsf{non}, \mathsf{acquire}, \mathsf{use}, \mathsf{release}\}$, which maps each context block to its corresponding resource interaction role. This labeling distinguishes blocks that manipulate resources from those that do not, enabling precise tracking of resource lifetimes and responsibilities.

```
try {
   NonResource nr1 = new ...;
   nr1.use();
   Resource r1 = new Resource();
   nr1.use();
   Resource r2 = new Resource(r1);
   r2.use();
   r2.close();
   nr1.use();
   r1.use();
   r1.close();
   nr1.use();
} catch (Exception e) {
   e.use();
}
```

Fig. 6: Example Code Used for Resource Flow Analysis

We model the method in Figure 6 as a sequence of labeled code blocks ($v_1 \sim v_{10}$), each annotated with a semantic tag τ that characterizes its resource-related behavior. The tags acquire, use, and non indicate resource acquisition, resource usage, and non-resource-related operations, respectively.

TABLE I: Labeled Code Blocks with Semantic Tags τ.

Node	Code Snippet	τ
v_1	`NonResource nr1 = new ...; nr1.use();`	non
v_2	`Resource r1 = new Resource();`	r1.acquire
v_3	`nr1.use();`	r1.use
v_4	`Resource r2 = new Resource(r1);`	r2.acquire
v_5	`r2.use(); r2.close();`	r2.release
v_6	`nr1.use();`	non
v_7	`r1.use(); r1.close();`	r1.release
v_8	`nr1.use();`	non
v_9	`catch (Exception e) {`	non
v_{10}	`e.use();`	non

Table I shows the annotated execution structure derived from context analysis of the example code. The execution is modeled as a set of semantically meaningful edges between code blocks v_1 through v_{10}. This structure separates the main execution sequence from exception-handling transitions. The edge categories are defined as follows:

- **Main Edge Sequential Flow:** $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_8$. This flow represents the normal execution order under non-exceptional conditions. It defines the core execution path and maintains the intended sequential semantics across blocks.
- **Sub Edge Flow (ex. Exceptional Flow):** $v_1 \rightarrow v_9$, $v_3 \rightarrow v_9$, $v_5 \rightarrow v_9$, $v_6 \rightarrow v_9$, $v_7 \rightarrow v_9$, $v_8 \rightarrow v_9$. These edges represent control transfers to the exception handler. Each edge originates from a basic block that may raise an exception during execution.
- **Sub Edge Flow (ex. Catch Block Flow):** $v_9 \rightarrow v_{10}$. This edge indicates the transition from the exception entry point to the corresponding catch handler. The catch handler executes only when an exception reaches block v_9.

The overall structure is organized around the Main Edge Flow, with additional sub-flows for exception propagation and handling. This abstraction not only preserves the structural execution semantics but also delineates resource-sensitive and resource-agnostic control paths with fine granularity. Such separation facilitates precise and minimally intrusive patch generation, ensuring that corrective transformations are contextually aligned while preserving the integrity of the original program logic.

*3) **Dependency Analysis**:* While Context Analysis identifies high-level regions such as `acquire` and `use`, it does not capture structural relationships between resource variables, including wrapping or interactions. These relationships are critical for enforcing correct release order and preventing misuse, such as use-after-close or double-close errors. To address this limitation, we construct a Resource Dependency Graph (RDG), a directed acyclic graph [31] $G_{\text{RDG}} = (V_R, E_R)$, where:

- V_R is the set of resource variables (e.g., `bos`, `oos`) extracted from context blocks.
- E_R is a set of directed edges (r_j, r_i) indicating that r_j depends on r_i and must be released afterward.

We infer E_R using lightweight, syntax-based rules that generalize across Java codebases. To avoid over-approximation, we add an edge only when clear syntactic or structural evidence supports the dependency. These rules are derived from an analysis of over 50 real-world resource leak cases from top-voted Java questions on StackOverflow [32]. The cases span domains such as I/O, networking, XML, and JDBC. From this analysis, we identify three recurring patterns:

- **Constructor-Passing (CP)**: r_i is passed into the constructor of r_j.
- **Method-Passing (MP)**: r_i is passed as an argument to a method invoked on r_j.
- **Return-Alias (RP)**: r_j is assigned the return value of a method call on r_i.

These rules operate purely intra-procedurally without requiring type resolution or interprocedural analysis, enabling scalable and efficient inference. For example, Table I shows that `r2` is instantiated in v_4 with `r1` passed to its constructor. The CP rule applies and adds the edge $(\texttt{r2}, \texttt{r1})$ to E_R, indicating that `r2` must be closed before `r1`. The patch synthesis engine enforces this by emitting `r2.close()` before `r1.close()`. Unlike prior approaches such as RLFixer [**?**] and LLM-based models, which often overlook such dependencies, our approach ensures semantic correctness. On the JLeaks benchmark, we estimated RDG preservation by applying our inference rules to multi-resource-leak cases and inspecting whether the synthesized patches respected the inferred release order. Amur preserved approximately 78% of RDG edges, demonstrating strong adherence to dependency constraints. In contrast, LLM-only baselines preserved only 41%, often violating essential close-order semantics. By focusing on intense and recurring patterns, Amur achieves high precision while maintaining correct release order.

C. Prompt-Based Patch Generation

In the final stage, Amur synthesizes patches using an LLM based on structured inputs derived from resource flow analysis. To effectively handle complex resource lifecycles, Amur employs a semantics-aware prompting strategy that encodes flow and dependency information in a format suitable for LLM processing. Specifically, Amur constructs the prompt $\mathcal{P}$ as a 6-tuple (3), capturing both structural and contextual aspects of resource usage.

$$\mathcal{P} = \left\langle \begin{array}{l} \texttt{JAVA\_CODE}, \\ \texttt{LEAK\_TYPE}, \\ \texttt{RESOURCE\_NAMES}, \\ \texttt{RESOURCE\_DEPENDENCIES}, \\ \texttt{CONTEXT\_BLOCKS}, \\ \texttt{EDGE\_RELATIONS} \end{array} \right\rangle \tag{3}$$

- `JAVA_CODE`: Full method body, providing lexical/syntactic context.
- `LEAK_TYPE`: Indicates single or multiple resource leaks.
- `RESOURCE_NAMES`: Names of leaked resources.
- `RESOURCE_DEPENDENCIES`: RDG edges (r_i, r_j) where r_i must be released before r_j.
- `CONTEXT_BLOCKS`: Semantically annotated code regions.
- `EDGE_RELATIONS`: CG edges (v_i, v_j) denoting valid control-flow transitions.

This structured prompt encodes behavioral and control dependencies, allowing the LLM to reason about resource lifecycles across basic blocks. By capturing block-level semantics and transitions, it enables precise, control-sensitive patch generation that adheres to resource management protocols

and remains effective even under complex control flow and dependency conditions.

```
### Instruction:
Fix the resource leak in the following Java
    method using the structured information
    below.

### Input:
JAVA_CODE: ...
LEAK_TYPE: multi
RESOURCE_NAMES: r1, r2
RESOURCE_DEPENDENCIES: [
  (r2, r1)  // r2 must be closed before r1
]
CONTEXT_BLOCKS: [
  {
    label: v1,
    role: non,
    dependency: [],
    snippet: "Non-Resource nr1 = new ...; nr1.
        use();"
  },
  {
    label: v2,
    role: acquire,
    dependency: [],
    snippet: "Resource r1 = new Resource();"
  },
  {
    label: v3,
    role: use,
    dependency: [],
    snippet: "nr1.use();"
  },
  {
    label: v4,
    role: acquire,
    dependency: [r1],
    snippet: "Resource r2 = new Resource(r1);"
  },
  {
    label: v5,
    role: use,
    dependency: [],
    snippet: "r2.use(); r2.close();"
  },
  ...
]
EDGE_RELATIONS: [
  (v1->v2), (v2->v3), (v3->v4), ...
  (v1->v9), (v3->v9), ...
]

### Output:
```

Fig. 7: Prompt construction based on resource flow analysis

Figure 7 shows a prompt constructed using the results of resource flow analysis (RFA) described in Sections III-B2 and III-B3. `CONTEXT_BLOCKS` correspond to nodes in the context graph $CG = (V, E, \tau)$, and `RESOURCE_DEPENDENCIES` encode release-order constraints extracted from the RDG. This structured format exposes control flow and data-flow semantics, enabling the LLM to understand resource behavior precisely. The prompt generated by Amur captures:

- Relevant and irrelevant operations concerning resource
- Inter-resource release constraints (e.g., $(\texttt{r2}, \texttt{r1})$),
- Usage continuity across control-flow paths.

D. Implementation Details

Amur comprises three components: static analysis, LLM-based patch generation, and reproducibility infrastructure. Each component functions independently and supports task-level reuse. For static analysis, Amur uses the `javalang` Python library [33] to parse Java syntax. A resource dictionary, derived from the official Java API [34], maps common resource types (e.g., `InputStream`, `Socket`) to their lifecycle methods such as constructors, `close()`, and `flush()`. For patch generation, Amur integrates the OpenAI API [35] and supports three GPT-based models: `GPT-4o`, `GPT-4o-mini`, and `o1`. These models are selected based on their strong performance on code-related tasks [36]–[39]. All inferences use temperature 0 for deterministic outputs [40], [41]. We conducted all experiments on Ubuntu 22.04 with an 8-core Intel i7 (3.2 GHz) and 32 GB RAM. Amur relies solely on standard Python libraries and an OpenAI API key, allowing for straightforward deployment. All experimental results are publicly available[1] to support reproducibility and future extensions.

IV. Research Questions

We evaluate the effectiveness and practicality of **Amur** by addressing the following questions:

- **RQ1:** How well does Amur repair *multi-resource leaks*?
  Method: Compare Amur with RLFixer and LLM baselines on the *JLeaks* dataset, measuring patch success and correctness.
- **RQ2:** How well does Amur handle *single-resource leaks*?
  Method: Evaluate Amur, RLFixer, and LLMs on *NJR-1* and *JLeaks*, measuring patch generation and correctness.
- **RQ3:** Does Amur succeed where LLMs fail?
  Method: Analyze failure cases involving complex control flow.
- **RQ4:** Do Amur's patches preserve program structure and control flow?
  Method: Measure token-level similarity and assess how RFA reduces unnecessary edits.

A. Experimental Setup

We evaluated **Amur** on two public benchmarks:

- **NJR-1** [13]: 2,205 Java methods with single-resource leaks, compiled from static analysis warnings used in RLFixer.
- **JLeaks** [28]: 715 analyzable cases (728 total, 13 excluded due to compilation errors), including both single- and multi-resource leaks with complex control structures.

[1]https://github.com/pinguskku/Amur

TABLE II: Multi-Resource Leak Patch Generation, Correctness, and Improvement Rates (JLeaks dataset).
*RLF: RLFixer baseline, *LLO: LLM-Only average

Approach	Gen.	Correctness	Correctness Gain
RLFixer	2%	0%	–
LLM-Only (GPT-4o)	92%	60%	+60% (RLF)
LLM-Only (GPT-4o-mini)	92%	60%	+60% (RLF)
LLM-Only (o1)	92%	61%	+61% (RLF)
LLM-Only (Avg)	**92%**	**60%**	**+60% (RLF)**
Amur (GPT-4o)	**96%**	**80%**	**+80% / +20% (RLF / LLO)**
Amur (GPT-4o-mini)	**96%**	**80%**	**+80% / +20% (RLF / LLO)**
Amur (o1)	**96%**	**70%**	**+70% / +10% (RLF / LLO)**
Amur (Avg)	**96%**	**76%**	**+76% / +16% (RLF / LLO)**

TABLE III: Single-Resource Leak Patch Generation, Correctness, and Improvement Rates (NJR-1 and JLeaks datasets).
*RLF: RLFixer baseline, *LLO: LLM-Only average

Approach	NJR-1		JLeaks		Total		Correctness Gain
	Gen.	Correctness	Gen.	Correctness	Gen.	Correctness	
RLFixer	66%	95%	4%	0%	35%	48%	–
LLM-Only (GPT-4o)	63%	57%	62%	62%	63%	60%	+12% (RLF)
LLM-Only (GPT-4o-mini)	63%	62%	62%	51%	63%	56%	+8% (RLF)
LLM-Only (o1)	63%	51%	62%	63%	63%	57%	+9% (RLF)
LLM-Only (Avg)	**63%**	**56%**	**62%**	**58%**	**62%**	**57%**	**+9% (RLF)**
Amur (GPT-4o)	**72%**	**80%**	**68%**	**88%**	**70%**	**84%**	**+36% / +27% (RLF / LLO)**
Amur (GPT-4o-mini)	**72%**	**76%**	**68%**	**83%**	**70%**	**80%**	**+32% / +23% (RLF / LLO)**
Amur (o1)	**72%**	**77%**	**68%**	**81%**	**70%**	**79%**	**+31% / +22% (RLF / LLO)**
Amur (Avg)	**72%**	**77%**	**68%**	**84%**	**70%**	**81%**	**+33% / +24% (RLF / LLO)**

We compared Amur against the following baselines:

- **RLFixer** [12]: A rule-based APR system targeting single-resource leaks.
- **LLM-Only**: GPT variants (`GPT-4o`, `GPT-4o-mini`, `o1`) prompted with raw buggy code without analysis context.

We used two evaluation metrics:

- **Generation Rate (%)**: Ratio of valid patches successfully generated.
- **Correctness Rate (%)**: Ratio of patches that eliminate the leak, preserve program semantics and introduce no new defects.

We applied a consistent three-stage evaluation pipeline: *sampling*, *static analysis filtering*, and *expert review*.

For **NJR-1**, following the RLFixer protocol, we randomly sampled 30 patches per static analysis tool, corresponding to 10% of the total patches.

For **JLeaks**, as no prior evaluation protocol exists, we performed stratified random sampling (10% per static analysis tool) while preserving the proportion of single- and multi-resource leaks.

All sampled patches were then analyzed using five static analysis tools: Infer, PMD, CF, CodeGuru, and SpotBugs. Only patches that passed all five analyzers without any warnings proceeded to the expert review stage. Two Java experts independently reviewed each patch and resolved disagreements through discussion, achieving an inter-rater agreement of 0.92 (Krippendorff's α [42]–[46]).

B. RQ1: Effectiveness on Multi-Resource Leak Repair

Table II summarizes the results of *multi-resource* leak repair. We compared **Amur** with two baselines on *multi-resource* leak repair: (1) **RLFixer**, a fixed-template RFA tool for single-resource leaks, and (2) **LLM-only** methods without static semantics. We evaluate (i) **generation rate**—whether the method produces a patch, and (ii) **correctness**—whether the patch eliminates the leak without introducing side effects. RLFixer, designed for single-resource cases, fails to produce any correct patches for multi-resource leaks. LLM-only methods produce patches in **92%** of cases, with **60–61%** correctness. Amur raises these to **96%** and **80%**, demonstrating clear benefits from static semantic guidance. Compared to RLFixer and LLM-only baselines, Amur improves correctness by **76%** and **16%**, respectively. All LLM variants showed similar trends, indicating that the improvement comes from Amur's synthesis strategy rather than model selection. These results confirm that static semantic guidance enables precise coordination across multiple resource instances, which is essential for sound multi-resource repair.

C. RQ2: Effectiveness on Single-Resource Leak Repair

Table III presents the results of Amur's single-resource leak repair on *NJR-1* and *JLeaks*, comparing patch generation and correctness against RLFixer and LLM-only baselines. **RLFixer**, while effective on NJR-1 (95% correctness), relies on rigid RFA templates and performs poorly on complex, real-world code in JLeaks (4% generation, 0% correctness). Its overall performance (35% generation, 48% correctness) highlights limited generalizability. **LLM-only** methods achieve 63% generation and 57% correctness but struggle without static semantic context. In contrast, **Amur** achieves 70% generation and 81% correctness—improving up to **33%** in generation and **24%** in correctness—demonstrating the effectiveness of static semantic guidance. The higher generation but slightly lower correctness on NJR-1 reflects Amur's broader patch exploration beyond RLFixer's conservative templates.

D. RQ3: Generalizability Across Complex Leak Scenarios

While RQ1 and RQ2 demonstrate Amur's advantage on standard metrics, they do not assess its ability to preserve critical semantic behavior. To address this, we analyze two complex leak scenarios from the JLeaks benchmark that involve realistic, semantics-sensitive patterns. We describe some examples of how RFA enables Amur to preserve meaningful program behavior that LLM-based baselines often miss.

```
try {
  PreparedStatement stat = new(...);
  ResultSet rs = stat.executeQuery();
  ...
  rs.close();
  stat.close();
} catch (SQLException e) {...}
```

Buggy Code (a)

```
try {
  ps = new(...);
  rs = ps.executeQuery();
  ...
} catch (Exception e) {...}
  finally {
  if (rs != null) rs.close();
  if (ps != null) ps.close();
}
```

Patch by LLM (b)

```
try (PreparedStatement ps = new(...);
     ResultSet rs = ps.executeQuery()) {
  ...
} catch (SQLException e) {...}
```

Patch by Amur (c)

Fig. 8: Case 1: Violation of Significant Semantics

Case 1: Preserving Exception Semantics: In Figure 8a, demonstrates a non-trivial resource leak scenario involving nested usage of `PreparedStatement` and `ResultSet` within a `try-catch` block. The original implementation explicitly catches `SQLException` to emit diagnostic messages upon close failures, thereby supporting transparent error reporting and preserving failure semantics critical to downstream logic. In Figure 8b, the patch generated by the LLM-only baseline (GPT-4o) introduces `finally` blocks for resource cleanup but omits the original exception logic, disrupting intended failure handling. As a result, the patch suppresses the intended diagnostic path and alters the observable failure behavior, introducing a semantic regression that could hinder debuggability and violate application-level error contracts. In Figure 8c, the Amur patch replaces the manual resource management logic with a `try`-with-resources construct while explicitly preserving the original `catch` clause. This ensures both resource safety and semantic equivalence with the original control-flow and exception behavior. Such preservation is not incidental but a direct consequence of Amur's static analysis of control and data dependencies. By explicitly modeling how exceptions propagate through the resource lifecycle, Amur identifies the functional significance of the `catch` block and ensures that the synthesized patch respects the original program's intent—something general-purpose LLMs fail to capture without semantic context.

```
InputStream input = new (...);
OutputStream output = new (...);
...
output.write(...);
output.flush();
output.close();
input.close();
init();
```

Buggy Code (a)

```
try (InputStream input = new (...);
     OutputStream output = new (...)) {
  ...
}
init();
```

Patch by LLM (b)

```
try (InputStream input = new (...);
     OutputStream output = new (...)) {
  ...
  output.flush();
}
init();
```

Patch by Amur (c)

Fig. 9: Case 2: Silent Removal of Significant Operation

Case 2: Preserving Semantically Significant Operations: In Figure 9a, the original method performs byte-stream I/O and explicitly invokes `flush()` before closing the output stream to ensure that all buffered data is reliably written. This pattern reflects a deliberate design choice for preserving data integrity, particularly in the presence of custom buffers or layered stream abstractions. In In Figure 9b, the LLM-only patch, generated

TABLE IV: Token and Control-Flow Structure Preservation for Semantically Correct Patches.

Approach	Token Preservation		CFG Similarity		Overall Avg.	
	Single	Multi	Single	Multi	Token	CFG
LLM-Only (GPT-4o)	72%	68%	65%	62%	70%	64%
LLM-Only (GPT-4o-mini)	74%	69%	64%	65%	72%	65%
LLM-Only (o1)	70%	71%	67%	63%	71%	65%
LLM-Only (avg)	**72%**	**69%**	**65%**	**63%**	**71%**	**65%**
Amur (GPT-4o)	**87%**	**84%**	**81%**	**78%**	**86%**	**80%**
Amur (GPT-4o-mini)	**85%**	**82%**	**80%**	**76%**	**84%**	**78%**
Amur (o1)	**83%**	**80%**	**77%**	**75%**	**82%**	**76%**
Amur (avg)	**85%**	**82%**	**79%**	**76%**	**84%**	**78%**

using GPT-4o, omits the `flush()` call and closes the stream directly, risking silent data loss and violating the intended reliability contract. In Figure 9c, the Amur patch retains the `flush()` invocation, guided by context-aware analysis that identifies the operation as semantically significant. This behavior illustrates Amur's ability to capture stream semantics and synthesize patches that maintain functional equivalence with the original code. Together with the first case Figure 8, this example highlights the advantage of Amur over LLM-only patches, which often overlook nuanced behaviors such as exception handling and side effect operations, leading to semantic regressions that compromise correctness despite syntactic validity.

Beyond the two case studies, we identified recurring failure patterns in patches generated by LLM-only baselines:

- **Misordered Resource Closure** — Closing dependent resources in an incorrect order (e.g., closing an inner compression stream before its outer wrapper), which breaks proper flushing semantics.
- **Removed Conditional Cleanup** — Eliminating conditionals guarding cleanup logic, resulting in the loss of behaviors such as selective release or failover handling.
- **Broken Control Flow** — Modifying or removing early `return` statements, which leads to incorrect execution paths or unreachable code.

Amur avoids these pitfalls by combining static analysis with structured prompting, enabling it to generate semantically correct patches that go beyond surface-level code fixes. Unlike LLM-only methods, Amur interprets low-level I/O operations in their broader semantic context, allowing it to preserve critical program behaviors such as reliable flushing, selective cleanup, and early returns. This capability proves essential for repairing real-world bugs where correctness depends not just on syntax but on adherence to implicit contracts and nuanced control flow semantics.

E. RQ4: Structural and Stylistic Fidelity Enabled by RFA

Beyond semantic correctness, a practical repair system must preserve the original code structure and style [47]–[51]. RFA enables this by generating patches guided by localized, context-aware resource-handling logic. We evaluate fidelity using two metrics: token-level similarity and control-flow consistency. The *Token Preservation Rate (TPR)* quantifies the percentage of original tokens retained in the patched method, defined (4):

$$\text{TPR} = \frac{|\mathcal{T}_{\text{orig}} \cap \mathcal{T}_{\text{patch}}|}{|\mathcal{T}_{\text{orig}}|} \times 100 \tag{4}$$

where $\mathcal{T}_{\text{orig}}$ and $\mathcal{T}_{\text{patch}}$ are the multisets of tokens from the original and patched methods. To assess structural consistency, we use *Control-Flow Graph Similarity (CFG-Sim)* [52], [53], computed via normalized graph edit distance.

As shown in Table IV, Amur achieves higher fidelity than all LLM-based baselines, with an average of 84% TPR and 78% CFG-Sim across correct patches from NJR-1 and JLeaks. In contrast, the strongest LLM-only baseline achieves only 71% TPR and 65% CFG-Sim. These results demonstrate that RFA enables minimal and stylistically coherent edits by restricting changes to semantically relevant regions and preserving control-flow structure.

```
InputStream is = getResourceAsStream(..);
if (is != null) {
  BufferedReader reader = new (..);
  String serviceName = reader.readLine();
  List<String> names = new ArrayList<>();
  while (serviceName != null && !"".equals(..)
      ) {
      ...
      serviceName = reader.readLine();
  }
  reader.close();
}
```

Fig. 10: Comparing Structural Preservation in Correct Repairs

Figure 10 compares the original code with two semantically correct patches that differ in structural and lexical fidelity.

The LLM-only patch introduces unnecessary edits that alter the original coding idioms:

- Moves `readLine()` into the `while` header, modifying the loop structure.
- Replaces `!"".equals(...)` with `!str.isEmpty()`, changing the original logic.
- Hoists `List<String> names` to an earlier scope than required.

Although functionally correct, these edits disregard the original structure and style, reducing code clarity and maintainability.

In contrast, the Amur patch performs minimal edits and preserves idiomatic usage:

- Maintains the original loop structure and placement of `readLine()`.
- Preserves the original logic and comparison idiom.
- Adds a `finally` block for cleanup without altering unrelated code.

By restricting edits to semantically relevant regions, Amur preserves both structural and lexical fidelity. We conducted a qualitative evaluation with five developers experienced in Java programming to assess patch preference. All participants consistently favored Amur patches over those from LLM-only baselines, citing improved control-flow preservation and stylistic fidelity. Inter-rater agreement was high ($\alpha = 0.88$), indicating strong consensus. These findings demonstrate that Amur generates correct and maintainable patches aligned with developer expectations.

V. Discussion

Resource leaks are inherently non-functional bugs, and no fully automated or precise method currently exists to ensure their complete detection or verification [54]–[60]. As a result, Amur employs manual inspection to assess the semantic correctness of its patches. Despite this limitation, Amur achieves strong inter-rater agreement, demonstrating consistent evaluation outcomes. Moreover, the substantial difference in patch quality with and without Amur reduces concerns about potential subjective bias. These results suggest that combining static reasoning with large language models offers a practical and effective strategy for addressing semantically critical repairs. Amur is also designed to be modular, making it compatible with future automated validation techniques and suitable as a foundation for more fully automated repair systems. Although some code patterns in JLeaks may appear similar to pre-training data issue [61]–[63], the significant performance gap between Amur and LLM only baselines, reaching up to 33% in correctness, indicates that the performance gain of Amur does not stem from memorization. If memorization were the primary factor, the LLM-only baselines would have achieved comparable results without additional guidance. Instead, the structured semantic inputs of Amur, including flow-aware block labeling and resource dependency graphs, substantially improve the reasoning capabilities of the LLM for resource usage, particularly in complex scenarios involving exceptions or multiple resources, where generalization is essential. To further understand the contribution of Amur's individual components, we conducted an ablation study. We found that using both the context graph and the resource dependency graph yields the best results. Specifically, removing the context graph led to a 16% decrease in accuracy, while removing the dependency graph caused a 21% drop, particularly in multi-resource scenarios. These findings indicate that both components are essential and complementary.

VI. Threats to Validity

Internal Validity: Patch correctness was evaluated manually based on predefined semantic criteria. Although the process involved multiple independent reviewers and yielded a high inter-rater agreement, some subjectivity may persist due to differences in reviewer expertise, particularly in cases involving subtle semantic behaviors.

External Validity: The evaluation was conducted on Java programs using two public benchmarks, NJR-1 and JLeaks. While JLeaks was designed to include diverse and realistic resource leak patterns, the results may not generalize to other programming languages, execution environments, or large-scale industrial systems. Further empirical studies are needed to assess the applicability of Amur across broader settings. In addition, while the findings in RQ3 and RQ4 are supported by representative case studies, these may not fully capture the range of real-world resource leak scenarios. Expanded quantitative evaluation would help strengthen the generalizability of our conclusions.

VII. Related Work

Resource Leak Detection: Static and dynamic tools such as Infer [64], PMD [65], SpotBugs [66], and the Checker Framework [67] detect unreleased resources through control-flow analysis. These tools are widely adopted in both industry and academic benchmarks like NJR-1 but focus solely on detection, offering no automated repair or reasoning about release orders. Amur extends this line of work by combining flow-sensitive detection with patch synthesis.

Automated Resource Leak Repair: Few APR systems directly target resource leaks. RLFixer [12] uses simple, single-purpose repair templates that work well for straightforward cases but struggle with complex control flows or multi-resource interactions due to their rule-based nature. Amur overcomes these limitations through a template-free approach that integrates static analysis with LLM-guided synthesis. InferFix [68] is another capable APR system, but it is tightly coupled with Infer and relies on Infer's specific bug reports and internal representations. This design limits its applicability in detector-agnostic settings and makes integration with alternative analysis pipelines difficult. Moreover, InferFix is primarily designed to operate in CI/CD pipelines within specific industrial toolchains rather than supporting diverse research or evaluation environments like Amur and RLFixer. These constraints make InferFix less suitable for general-purpose, modular repair workflows.

Semantics-Aware Program Repair: Semantic APR techniques [69]–[72] leverage symbolic reasoning or learned representations to preserve correctness, primarily focusing on logic bugs. In contrast, non-functional issues [73], [74] such as resource misuse remain underexplored. Amur introduces semantics-aware repair into this domain by modeling control flow, contextual interactions, and inter-resource dependencies to synthesize behavior-preserving patches with LLM guidance.

VIII. Conclusion

Resource leaks remain a prevalent and costly class of bugs, particularly when multiple resources interact under complex control flows. While large language models (LLMs) show promise for automated code repair, they often fail to preserve subtle semantics and developer intent in resource management. We presented **Amur**, a semantics-aware context synthesis approach that guides LLMs in generating correct patches for resource leaks. Amur analyzes flow-sensitive resource usage and inter-resource dependencies through lightweight static analysis, then constructs structured prompts that encode this semantic context. These prompts enable the LLM to produce patches that are both context-aware and behavior-preserving. Experiments on NJR-1 and JLeaks demonstrate that Amur improves patch correctness, reduces semantic regressions, and better preserves program structure compared to template-based and LLM-only baselines. Although Amur currently targets Java and intra-procedural repair, the results highlight the value of combining static reasoning with generative models. Future work will extend Amur to support asynchronous resource protocols, interprocedural reasoning, and broader validation across languages and industrial-scale codebases.

Acknowledgment

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00438686, Development of software reliability improvement technology through identification of abnormal open sources and automatic application of DevSecOps).

References

[1] MITRE, "Common Weakness Enumeration (CWE-400): Uncontrolled Resource Consumption." https://cwe.mitre.org/data/definitions/400.html, 2022. Accessed: 2025-03-15.

[2] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 535–544, 2010.

[3] Z. Dai, X. Mao, Y. Lei, X. Wan, and K. Ben, "Resco: Automatic collection of leaked resources," *IEICE TRANSACTIONS on Information and Systems*, vol. 96, no. 1, pp. 28–39, 2013.

[4] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak, "Memory and resource leak defects and their repairs in java projects," vol. 25, pp. 678–718, Springer, 2020.

[5] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pp. 298–309, 2018.

[6] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[7] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," *arXiv preprint arXiv:2305.04087*, 2023.

[8] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*, pp. 65–86, Springer, 2018.

[9] Z. Xu, Q. Li, and S. H. Tan, "Guiding chatgpt to fix web ui tests via explanation-consistency checking," *arXiv preprint arXiv:2312.05778*, 2023.

[10] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th international conference on software engineering*, pp. 1506–1518, 2022.

[11] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "Energypatch: Repairing resource leaks to improve energy-efficiency of android apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 470–490, 2017.

[12] A. Utture and J. Palsberg, "From leaks to fixes: Automated repairs for resource leak warnings," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 159–171, 2023.

[13] A. Utture, C. G. Kalhauge, S. Liu, and J. Palsberg, "Njr-1 dataset," 2020.

[14] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[15] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, 2024.

[16] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2332–2354, 2024.

[17] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 21841–21849, 2024.

[18] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pp. 1–6, 2024.

[19] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," in *Proceedings of the 46th ieee/acm international conference on software engineering*, pp. 1–13, 2024.

[20] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, and F. Liu, "Enhancing automated program repair with solution design," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1706–1718, 2024.

[21] Y. Li, F. H. Shezan, B. Wei, G. Wang, and Y. Tian, "Sok: Towards effective automated vulnerability repair," *arXiv preprint arXiv:2501.18820*, 2025.

[22] C. Munley, A. Jarmusch, and S. Chandrasekaran, "Llm4vv: Developing llm-driven testsuite for compiler validation," *Future Generation Computer Systems*, vol. 160, pp. 1–13, 2024.

[23] L. Albuquerque and R. Gheyi, "Investigating llm capabilities in the identification of compilation errors in configurable systems," in *Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*, pp. 39–48, SBC, September 2024.

[24] A. Taylor, A. Vassar, J. Renzella, and H. Pearce, "Dcc–help: Transforming the role of the compiler by generating context-aware error explanations with large language models," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pp. 1314–1320, ACM, March 2024.

[25] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Llm compiler: Foundation language models for compiler optimization," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, pp. 141–153, ACM, February 2025.

[26] P. Widjojo and C. Treude, "Addressing compiler errors: Stack overflow or large language models?," *arXiv preprint*, vol. arXiv:2307.10793, 2023.

[27] D. Garmus and D. Herron, *Function point analysis: measurement practices for successful software projects*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[28] T. Liu, W. Ji, X. Dong, W. Yao, Y. Wang, H. Liu, H. Peng, and Y. Wang, "Jleaks: A featured resource leak repository collected from hundreds of open-source java projects," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[29] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pp. 31–42, 2019.

[30] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1274–1286, ACM, September 2024.

[31] S. Overflow, "Why leak occurs in retain-cycle structure?." https://stackoverflow.com/questions/791322/

retain-cycles-why-is-that-such-a-bad-thing, 2025. Accessed: 2025-03-15.
[32] S. Overflow, "Java tag on stack overflow." https://stackoverflow.com/questions/tagged/java, 2025. Accessed: 2025-03-15.
[33] N. Smith, "javalang: Pure python java parser and tools." https://github.com/c2nes/javalang, 2016. Accessed: 2025-03-15.
[34] oracle, "Java documentation." https://docs.oracle.com/javase/8/docs/api/, 2025. Accessed: 2025-03-15.
[35] OpenAI, "Openai api." https://openai.com/index/openai-api/, 2023. Accessed: 2025-03-15.
[36] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pp. 103–111, ACM, July 2024.
[37] T. Ahmed and P. Devanbu, "Better patching using llm prompting, via self-consistency," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1742–1746, IEEE, September 2023.
[38] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "Automated software vulnerability patching using large language models," *arXiv preprint arXiv:2408.13597*, 2024.
[39] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1646–1656, ACM, November 2023.
[40] M. Rosoł, J. S. Gasior, J. Łaba, K. Korzeniewski, and M. Młyńczak, "Evaluation of the performance of gpt-3.5 and gpt-4 on the polish medical final examination," *Scientific Reports*, vol. 13, no. 1, p. 20512, 2023.
[41] M. Mitchell, A. B. Palmarini, and A. Moskvichev, "Comparing humans, gpt-4, and gpt-4v on abstraction and reasoning tasks," *arXiv preprint arXiv:2311.09247*, 2023.
[42] X. Zhao, G. C. Feng, J. S. Liu, and K. Deng, "We agreed to measure agreement-redefining reliability de-justifies krippendorff's alpha.," *China media research*, vol. 14, no. 2, 2018.
[43] K. Krippendorff, "Computing krippendorff's alpha-reliability," 2011.
[44] J. Hughes, "krippendorffsalpha: An r package for measuring agreement using krippendorff's alpha coefficient," *arXiv preprint arXiv:2103.12170*, 2021.
[45] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Communication methods and measures*, vol. 1, no. 1, pp. 77–89, 2007.
[46] K. Krippendorff, "Agreement and information in the reliability of coding," *Communication methods and measures*, vol. 5, no. 2, pp. 93–112, 2011.
[47] F. N. Meem, J. Smith, and B. Johnson, "Exploring experiences with automated program repair in practice," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–11, 2024.
[48] Y. Zhang, Z. Jin, Y. Xing, G. Li, F. Liu, J. Zhu, W. Dou, and J. Wei, "Patch: Empowering large language model with programmer-intent guidance and collaborative-behavior simulation for automatic bug fixing," *ACM Transactions on Software Engineering and Methodology*, 2025.
[49] Z. Chen and L. Jiang, "Evaluating software development agents: Patch patterns, code quality, and issue complexity in real-world github scenarios," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 657–668, IEEE, 2025.
[50] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 271–280, IEEE, 2014.
[51] Z. Fei, J. Ge, C. Li, T. Wang, Y. Li, H. Zhang, L. Huang, and B. Luo, "Patch correctness assessment: A survey," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–50, 2025.
[52] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and applications*, vol. 13, pp. 113–129, 2010.
[53] Z. Zeng, A. K. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 25–36, 2009.
[54] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in android applications," in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pp. 176–183, IEEE, 2014.
[55] M. Ghanavati and A. Andrzejak, "Automated memory leak diagnosis by regression testing," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 191–200, IEEE, 2015.
[56] D. Amalfitano, V. Riccio, P. Tramontana, and A. R. Fasolino, "Do memories haunt you? an automated black box testing approach for detecting memory leaks in android apps," *IEEE Access*, vol. 8, pp. 12217–12231, 2020.
[57] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 115–125, 2005.
[58] D. Yan, G. Xu, S. Yang, and A. Rountev, "Leakchecker: Practical static memory leak detection for managed languages," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 87–97, 2014.
[59] M. Li, Y. Chen, L. Wang, and G. Xu, "Dynamically validating static memory leak warnings," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 112–122, 2013.
[60] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pp. 156–164, 2004.
[61] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, "Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 642–646, 2024.
[62] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pp. 103–111, 2024.
[63] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "Automated software vulnerability patching using large language models," *arXiv preprint arXiv:2408.13597*, 2024.
[64] F. AI, "Infer: A static analysis tool for java, c, c++, and objective-c." https://github.com/facebook/infer, 2015. Accessed: 2025-03-15.
[65] P. Developers, "Pmd: Source code analyzer for java, javascript, apex, and more." https://github.com/pmd/pmd, 2025. Accessed: 2025-03-15.
[66] S. Developers, "Spotbugs: Static analysis tool for java programs." https://github.com/spotbugs/spotbugs, 2025. Accessed: 2025-03-15.
[67] C. F. Developers, "Checker framework: Pluggable type-checking for java." https://github.com/typetools/checker-framework, 2025. Accessed: 2025-03-15.
[68] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1646–1656, 2023.
[69] S. Mechtaev, M. D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 129–139, ACM, May 2018.
[70] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781, IEEE, May 2013.
[71] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. L. Goues, "Sosrepair: Expressive semantic search for real-world program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2162–2181, 2019.
[72] X.-B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 163–163, ACM, May 2018.
[73] A. Radu and S. Nadi, "A dataset of non-functional bugs," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 399–403, IEEE, 2019.
[74] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *2013 10th working conference on mining software repositories (MSR)*, pp. 237–246, IEEE, 2013.